手把手教你做一个 C 语言编译器

wizardforcel



Published with GitBook

目錄

手把手教你做一个 C 语言编译器

来源：

1. 手把手教你做一个 C 语言编译器（0）：前言
2. 手把手教你做一个 C 语言编译器（1）：设计
3. 手把手教你做一个 C 语言编译器（2）：虚拟机
4. 手把手教你做一个 C 语言编译器（3）：词法分析器
5. 手把手教你做一个 C 语言编译器（4）：递归下降
6. 手把手教你做一个 C 语言编译器（5）：变量定义
7. 手把手教你做一个 C 语言编译器（6）：函数定义
8. 手把手教你做一个 C 语言编译器（7）：语句
9. 手把手教你做一个 C 语言编译器（8）：表达式
10. 手把手教你做一个 C 语言编译器（9）：总结

手把手教你做一个 C 语言编译器（0）：前言

“手把手教你构建 C 语言编译器” 这一系列教程将带你从头编写一个 C 语言的编译器。希望通过这个系列，我们能对编译器的构建有一定的了解，同时，我们也将构建出一个能用的 C 语言编译器，尽管有许多语法并不支持。

在开始进入正题之前，本篇是一些闲聊，谈谈这个系列的初衷。如果你急切地想进入正篇，请跳过本章。

前言

为什么要学编译原理

如果要我说计算机专业最重要的三门课，我会说是《数据结构》、《算法》和《编译原理》。在我看来，能不能理解“递归”像是程序员的第一道门槛，而会不会写编译器则是第二道。

（当然，并不是说是没写过编译器就不是好程序员，只能说它是一个相当大的挑战吧）

以前人们会说，学习了编译原理，你就能写出更加高效的代码，但随着计算机性能的提升，代码是否高效显得就不那么重要了。那么为什么要学习编译原理呢？

原因只有一个：装B。

好吧，也许现在还想学习编译原理的人只可能是因为兴趣了。一方面想了解它的工作原理；另一方面希望挑战一下自己，看看自己能走多远。

理论很复杂，实现也很复杂？

我对编译器一直心存敬佩。所以当学校开《编译原理》的课程后，我是抱着满腔热情去上课的，但是两节课后我就放弃了。原因是太复杂了，听不懂。

一般编译原理的课程会说一些：

1. 如何表示语法（BNF什么的）
2. 词法分析，用什么有穷自动机和无穷自动机
3. 语法分析，递归下降法，什么 `LL(k)` ，LALR 分析。
4. 中间代码的表示
5. 代码的生成
6. 代码优化

我相信绝大多数（98%）的学生顶多学到语法分析就结束了。并且最重要的是，学了这么多也没用！依旧帮助不了我们学习编译器！这其中最主要的原因是《编译原理》试图教会我们的是如何构造“编译器生成器”，即构造一个工具，根据文法来生成编译器（如 lex/yacc）等等。

这些理论试图教会我们如何用通用的方法来自动解决问题，它们有很强的实际意义，只是对于一般的学生或程序员来说，它们过于强大，内容过于复杂。如果你尝试阅读 lex/yacc （或 flex/bison）的代码，就会发现太可怕了。

然而如果你能跟我一样，真正来实现一个简单的编译器，那么你会发现，比起可怕的《编译原理》，这点复杂度还是不算什么的（因为好多理论根本用不上）。

项目的初衷

有一次在 Github 上看到了一个项目（当时很火的），名叫 c4，号称用 4 个函数来实现了一个小的 C 语言编译器。它最让我震惊的是能够自举，即能自己编译自己。并且它用很少的代码就完成了一个功能相当完善的 C 语言编译器。

一般的编译器相关的教程要么就十分简单（如实现四则运算），要么就是借助了自动生成的工具（如 flex/bison）。而 c4 的代码完全是手工实现的，不用外部工具。可惜的是它的代码初衷是代码最小化，所以写得很乱，很难懂。所以本项目的主要目的：

1. 实现一个功能完善的 C 语言编译器
2. 通过教程来说明这个过程。

c4 大致500+行。重写的代码历时一周，总共代码加注释1400行。项目地址: Write a C Interpreter。

声明：本项目中的代码逻辑绝大多数取自 c4 ，但确为自己重写。

预警

在写编译器的时候会遇到两个主要问题：

1. 麻烦，会有许多类似的代码，写起来很无聊。
2. 难以调试，一方面没有很好的测试用例，另一方面需要对照生成的代码来调试（遇到的时候就知道了）。

所以我希望你有足够的耐心和时间来学习，相信当你真正完成的时候会像我一样，十分有成就感。

PS. 第一篇完全没有正题相关的内容也是希望你能有所心理准备再开始学习。

参考资料

最后想介绍几个资料：

1. Let’s Build a Compiler 很好的初学者教程，英文的。
2. Lemon Parser Generator，一个语法分析器生成器，对照《编译原理》观看效果更佳。

祝你学得愉快。

手把手教你做一个 C 语言编译器（1）：设计

本章是“手把手教你构建 C 语言编译器”系列的第二篇，我们要从整体上讲解如何设计我们的 C 语言编译器。

本系列：

1. 手把手教你做一个 C 语言编译器（0）：前言

首先要说明的是，虽然标题是编译器，但实际上我们构建的是 C 语言的解释器，这意味着我们可以像运行脚本一样去运行 C 语言的源代码文件。这么做的理由有两点：

1. 解释器与编译器仅在代码生成阶段有区别，而其它方面如词法分析、语法分析是一样的。
2. 解释器需要我们实现自己的虚拟机与指令集，而这部分能帮助我们了解计算机的工作原理。

编译器的构建流程

一般而言，编译器的编写分为 3 个步骤：

1. 词法分析器，用于将字符串转化成内部的表示结构。
2. 语法分析器，将词法分析得到的标记流（token）生成一棵语法树。
3. 目标代码的生成，将语法树转化成目标代码。

已经有许多工具能帮助我们处理阶段1和2，如 flex 用于词法分析，bison 用于语法分析。只是它们的功能都过于强大，屏蔽了许多实现上的细节，对于学习构建编译器帮助不大。所以我们要完全手写这些功能。

所以我们会根据下面的流程：

1. 构建我们自己的虚拟机以及指令集。这后生成的目标代码便是我们的指令集。
2. 构建我们的词法分析器
3. 构建语法分析器

编译器的框架

我们的编译器主要包括 4 个函数：

1. `next()` 用于词法分析，获取下一个标记，它将自动忽略空白字符。
2. `program()` 语法分析的入口，分析整个 C 语言程序。
3. `expression(level)` 用于解析一个表达式。
4. `eval()` 虚拟机的入口，用于解释目标代码。

这里有一个单独用于解析“表达式”的函数 `expression` 是因为表达式在语法分析中相对独立并且比较复杂，所以我们将它单独作为一个模块（函数）。

因为我们的源代码看起来就像是：

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <string.h>

int token; // current token
char *src, *old_src; // pointer to source code string;
int poolsize; // default size of text/data/stack
int line; // line number

void next() {
token = *src++;
return;
}

void expression(int level) {
// do nothing
}

void program() {
next(); // get next token
while (token > 0) {
printf("token is: %c\n", token);
next();
}
}

int eval() { // do nothing yet
return 0;
}

int main(int argc, char **argv)
{
int i, fd;

argc--;
argv++;

poolsize = 256 * 1024; // arbitrary size
line = 1;

if ((fd = open(*argv, 0)) < 0) {
printf("could not open(%s)\n", *argv);
return -1;
}

if (!(src = old_src = malloc(poolsize))) {
```

```
printf("could not malloc(%d) for source area\n", poolsize);
return -1;
}

// read the source file
if ((i = read(fd, src, poolsize-1)) <= 0) {
printf("read() returned %d\n", i);
return -1;
}
src[i] = 0; // add EOF character
close(fd);

program();
return eval();
}
```

上面的代码看上去挺复杂，但其实内容不多，就是读取一个源代码文件，逐个读取每个字符，并输出每个字符。这里重要的是注意每个函数的作用，后面的文章中，我们将逐个填充每个函数的功能，最终构建起我们的编译器。

本节的代码可以在 Github 上下载，也可以直接 clone

```
git clone -b step-0 https://github.com/lotabout/write-a-C-interpret
```

这样我们就有了一个最简单的编译器：什么都不干的编译器，下一章中，我们将实现其中的 `eval` 函数，即我们自己的虚拟机。

手把手教你做一个 C 语言编译器（2）：虚拟机

本章是“手把手教你构建 C 语言编译器”系列的第三篇，本章我们要构建一台虚拟的电脑，设计我们自己的指令集，运行我们的指令集，说得通俗一点就是自己实现一套汇编语言。它们将作为我们的编译器最终输出的目标代码。

本系列：

1. 手把手教你做一个 C 语言编译器（0）：前言
2. 手把手教你做一个 C 语言编译器（1）：设计

计算机的内部工作原理

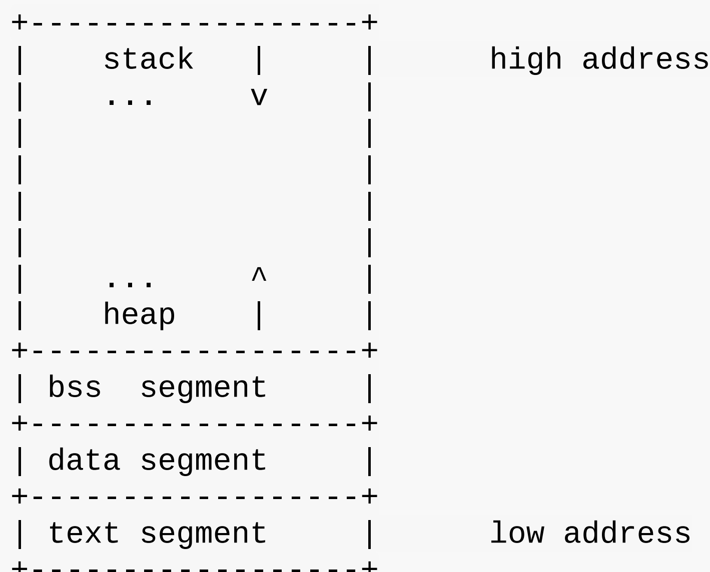
我们关心计算机的三个基本部件：CPU、寄存器及内存。代码（汇编指令）以二进制的形式保存在内存中，CPU 从中一条条地加载指令执行。程序运行的状态保存在寄存器中。

内存

我们从内存开始说起。现代的操作系统都不直接使用内存，而是使用虚拟内存。虚拟内存可以理解为一种映射，在我们的程序眼中，我们可以使用全部的内存地址，而操作系统需要将它映射到实际的内存上。当然，这些并不重要，重要的是一般而言，进程的内存会被分成几个段：

1. 代码段（text）用于存放代码（指令）。
2. 数据段（data）用于存放初始化了的数据，如 `int i = 10;` ，就需要存放到数据段中。
3. 未初始化数据段（bss）用于存放未初始化的数据，如 `int i[1000];` ，因为不关心其中的真正数值，所以单独存放可以节省空间，减少程序的体积。
4. 栈（stack）用于处理函数调用相关的数据，如调用帧（calling frame）或是函数的局部变量等。
5. 堆（heap）用于为程序动态分配内存。

它们在内存中的位置类似于下图：



但我们的虚拟机并不模拟完整的计算机，我们只关心三个内容：代码段、数据段以及栈。其中的数据段我们只存放字符串，因为我们的编译器并不支持初始化变量，因此我们也不需要未初始化数据段。理论上我们的虚拟器需要维护自己的堆用于内存分配，但实际实现上较为复杂且与编译无关，故我们引入一个指令 `MSET` ，使我们能直接使用编译器（解释器）中的内存。

综上，我们需要首先在全局添加如下代码：

```
int *text, // text segment
*old_text, // for dump text segment
*stack; // stack
char *data; // data segment
```

注意这里的类型，虽然是 `int` 型，但理解起来应该作为无符号的整型，因为我们会在代码段（text）中存放如指针/内存地址的数据，它们就是无符号的。其中数据段（data）由于只存放字符串，所以是 `char *` 型的

接着，在 `main` 函数中加入初始化代码，真正为其分配内存：

```
int main() {
close(fd);
...

// allocate memory for virtual machine
if (!(text = old_text = malloc(poolsize))) {
printf("could not malloc(%d) for text area\n", poolsize);
return -1;
}
if (!(data = malloc(poolsize))) {
printf("could not malloc(%d) for data area\n", poolsize);
return -1;
}
if (!(stack = malloc(poolsize))) {
printf("could not malloc(%d) for stack area\n", poolsize);
return -1;
}

memset(text, 0, poolsize);
memset(data, 0, poolsize);
memset(stack, 0, poolsize);

...
program();
```

寄存器

计算机中的寄存器用于存放计算机的运行状态，真正的计算机中有许多不同种类的寄存器，但我们的虚拟机中只使用 4 个寄存器，分别如下：

1. `PC` 程序计数器，它存放的是一个内存地址，该地址中存放着 下一条 要执行的计算机指令。
2. `SP` 指针寄存器，永远指向当前的栈顶。注意的是由于栈是位于高地址并向低地址增长的，所以入栈时 `SP` 的值减小。
3. `BP` 基址指针。也是用于指向栈的某些位置，在调用函数时会使用到它。
4. `AX` 通用寄存器，我们的虚拟机中，它用于存放一条指令执行后的结果。

要理解这些寄存器的作用，需要去理解程序运行中会有哪些状态。而这些寄存器只是用于保存这些状态的。

在全局中加入如下定义：

```
int *pc, *bp, *sp, ax, cycle; // virtual machine registers
```

在 `main` 函数中加入初始化代码，注意的是`PC`在初始应指向目标代码中的 `main` 函数，但我们还没有写任何编译相关的代码，因此先不处理。代码如下：

```
memset(stack, 0, poolsize);
...

bp = sp = (int *)((int)stack + poolsize);
ax = 0;

...
program();
```

与 CPU 相关的是指令集，我们将专门作为一个小节。

指令集

指令集是 CPU 能识别的命令的集合，也可以说是 CPU 能理解的语言。这里我们要为我们的虚拟机构建自己的指令集。它们基于 x86 的指令集，但要更为简单。

首先在全局变量中加入一个枚举类型，这是我们要支持的全部指令：

```
// instructions
enum { LEA ,IMM ,JMP ,CALL,JZ  ,JNZ ,ENT ,ADJ ,LEV ,LI  ,LC  ,SI  ,
OR  ,XOR ,AND ,EQ  ,NE  ,LT  ,GT  ,LE  ,GE  ,SHL ,SHR ,ADD ,SUB ,MU
OPEN,READ,CLOS,PRTF,MALC,MSET,MCMP,EXIT };
```

这些指令的顺序安排是有意的，稍后你会看到，带有参数的指令在前，没有参数的指令在后。这种顺序的唯一作用就是在打印调试信息时更加方便。但我们讲解的顺序并不依据它。

MOV

`MOV` 是所有指令中最基础的一个，它用于将数据放进寄存器或内存地址，有点类似于 C 语言中的赋值语句。x86 的 `MOV` 指令有两个参数，分别是源地址和目标地址：`MOV dest, source` （Intel 风格），表示将 `source` 的内容放在 `dest` 中，它们可以是一个数、寄存器或是一个内存地址。

一方面，我们的虚拟机只有一个寄存器，另一方面，识别这些参数的类型（是数还是地址）是比较困难的，因此我们将 `MOV` 指令拆分成 5 个指令，这些指令只接受一个参数，如下：

1. IMM <num> 将 <num> 放入寄存器 `ax` 中。
2. `LC` 将对应地址中的字符载入 `ax` 中，要求 `ax` 中存放地址。
3. `LI` 将对应地址中的整数载入 `ax` 中，要求 `ax` 中存放地址。
4. `SC` 将 `ax` 中的数据作为字符存放入地址中，要求栈顶存放地址。
5. `SI` 将 `ax` 中的数据作为整数存放入地址中，要求栈顶存放地址。

你可能会觉得将一个指令变成了许多指令，整个系统就变得复杂了，但实际情况并非如此。首先是 `MOV` 指令其实有许多变种，根据类型的不同有 `MOVB` , `MOVW` 等指令，我们这里的 `LC/SC` 和 `LI/SI` 就是对应字符型和整型的存取操作。

但最为重要的是，通过将 `MOV` 指令拆分成这些指令，只有 `IMM` 需要有参数，且不需要判断类型，所以大大简化了实现的难度。

在 `eval()` 函数中加入下列代码：

```
void eval() {
int op, *tmp;
while (1) {
if (op == IMM)       {ax = *pc++;} // load immediate value to ax
else if (op == LC)   {ax = *(char *)ax;} // load character to ax, a
else if (op == LI)   {ax = *(int *)ax;} // load integer to ax, addr
else if (op == SC)   {ax = *(char *)*sp++ = ax;} // save character
else if (op == SI)   {*(int *)*sp++ = ax;} // save integer to addre
}

...
return 0;
}
```

其中的 `*sp++` 的作用是退栈，相当于 `POP` 操作。

这里要解释的一点是，为什么 `SI/SC` 指令中，地址存放在栈中，而 `LI/LC` 中，地址存放在 `ax` 中？原因是默认计算的结果是存放在 `ax` 中的，而地址通常是需要通过计算获得，所以执行 `LI/LC` 时直接从 `ax` 取值会更高效。另一点是我们的 `PUSH` 指令只能将 `ax` 的值放到栈上，而不能以值作为参数，详细见下文。

PUSH

在 x86 中， `PUSH` 的作用是将值或寄存器，而在我们的虚拟机中，它的作用是将 `ax` 的值放入栈中。这样做的主要原因是为了简化虚拟机的实现，并且我们也只有一个寄存器 `ax` 。代码如下：

```
else if (op == PUSH) {*--sp = ax;} // push the value of ax onto the
```

JMP

`JMP <addr>` 是跳转指令，无条件地将当前的 `PC` 寄存器设置为指定的 `<addr>` ，实现如下：

```
else if (op == JMP)  {pc = (int *)*pc;} // jump to the address
```

要记得，`pc` 寄存器指向的是 下一条 指令。所以此时它存放的是 `JMP` 指令的参数，即 `<addr>` 的值。

JZ/JNZ

为了实现 `if` 语句，我们需要条件判断相关的指令。这里我们只实现两个最简单的条件判断，即结果（`ax`）为零或不为零情况下的跳转。

实现如下：

```
else if (op == JZ)   {pc = ax ? pc + 1 : (int *)*pc;} // jump if a
```

```
else if (op == JNZ)  {pc = ax ? (int *)*pc : pc + 1;} // jump if a
```

子函数调用

这是汇编中最难理解的部分，所以合在一起说，要引入的命令有 `CALL`，`ENT`，`ADJ` 及 `LEV`。

首先我们介绍 `CALL <addr>` 与 `RET` 指令，`CALL` 的作用是跳转到地址为 `<addr>` 的子函数，`RET` 则用于从子函数中返回。

为什么不能直接使用 `JMP` 指令呢？原因是当我们从子函数中返回时，程序需要回到跳转之前的地方继续运行，这就需要事先将这个位置信息存储起来。反过来，子函数要返回时，就需要获取并恢复这个信息。因此实际中我们将 `PC` 保存在栈中。如下：

```
else if (op == CALL) {*--sp = (int)(pc+1); pc = (int *)*pc;} // ca
//else if (op == RET)  {pc = (int *)*sp++;}
```

这里我们把 `RET` 相关的内容注释了，是因为之后我们将用 `LEV` 指令来代替它。

在实际调用函数时，不仅要考虑函数的地址，还要考虑如何传递参数和如何返回结果。这里我们约定，如果子函数有返回结果，那么就在返回时保存在 `ax` 中，它可以是一个值，也可以是一个地址。那么参数的传递呢？

各种编程语言关于如何调用子函数有不同的约定，例如 C 语言的调用标准是：

1. 由调用者将参数入栈。
2. 调用结束时，由调用者将参数出栈。
3. 参数逆序入栈。

事先声明一下，我们的编译器参数是顺序入栈的，下面的例子（C 语言调用标准）取自 维基百科：

```
int callee(int, int, int);

int caller(void)
{
int i, ret;

ret = callee(1, 2, 3);
ret += 5;
return ret;
}
```

会生成如下的 x86 汇编代码：

```
caller:
; make new call frame
push    ebp
mov     ebp, esp
sub     1, esp       ; save stack for variable: i
; push call arguments
push    3
push    2
push    1
; call subroutine 'callee'
call    callee
; remove arguments from frame
add     esp, 12
; use subroutine result
add     eax, 5
; restore old call frame
mov     esp, ebp
pop     ebp
; return
ret
```

上面这段代码在我们自己的虚拟机里会有几个问题：

1. `push ebp` ，但我们的 `PUSH` 指令并无法指定寄存器。
2. `mov ebp, esp` ，我们的 `MOV` 指令同样功能不足。
3. `add esp, 12` ，也是一样的问题（尽管我们还没定义）。

也就是说由于我们的指令过于简单（如只能操作 `ax` 寄存器），所以用上面提到的指令，我们连函数调用都无法实现。而我们又不希望扩充现有指令的功能，因为这样实现起来就会变得复杂，因此我们采用的方法是增加指令集。毕竟我们不是真正的计算机，增加指令会消耗许多资源（钱）。

ENT

`ENT <size>` 指的是 `enter`，用于实现 'make new call frame' 的功能，即保存当前的栈指针，同时在栈上保留一定的空间，用以存放局部变量。对应的汇编代码为：

```
; make new call frame
push    ebp
mov     ebp, esp
sub     1, esp       ; save stack for variable: i
```

实现如下：

```
else if (op == ENT)  {*--sp = (int)bp; bp = sp; sp = sp - *pc++;} /
```

ADJ

`ADJ <size>` 用于实现 'remove arguments from frame'。在将调用子函数时压入栈中的数据清除，本质上是因为我们的 `ADD` 指令功能有限。对应的汇编代码为：

```
; remove arguments from frame
add     esp, 12
```

实现如下：

```
else if (op == ADJ)  {sp = sp + *pc++;} // add esp, <size>
```

LEV

本质上这个指令并不是必需的，只是我们的指令集中并没有 `POP` 指令。并且三条指令写来比较麻烦且浪费空间，所以用一个指令代替。对应的汇编指令为：

```
; restore old call frame
mov     esp, ebp
pop     ebp
; return
ret
```

具体的实现如下：

```
else if (op == LEV)  {sp = bp; bp = (int *)*sp++; pc = (int *)*sp+
```

注意的是，`LEV` 已经把 `RET` 的功能包含了，所以我们不再需要 `RET` 指令。

LEA

上面的一些指令解决了调用帧的问题，但还有一个问题是如何在子函数中获得传入的参数。这里我们首先要了解的是当参数调用时，栈中的调用帧是什么样的。我们依旧用上面的例子（只是现在用“顺序”调用参数）：

```
sub_function(arg1, arg2, arg3);

|    ....       | high address
+---------------+
| arg: 1        |    new_bp + 4
+---------------+
| arg: 2        |    new_bp + 3
+---------------+
| arg: 3        |    new_bp + 2
+---------------+
|return address |    new_bp + 1
+---------------+
| old BP        | <- new BP
+---------------+
| local var 1   |    new_bp - 1
+---------------+
| local var 2   |    new_bp - 2
+---------------+
|    ....       |  low address
```

所以为了获取第一个参数，我们需要得到 `new_bp + 4`，但就如上面的说，我们的 `ADD` 指令无法操作除 `ax` 外的寄存器，所以我们提供了一个新的指令：`LEA <offset>`

实现如下：

```
else if (op == LEA)  {ax = (int)(bp + *pc++);} // load address for
```

以上就是我们为了实现函数调用需要的指令了。

运算符指令

我们为 C 语言中支持的运算符都提供对应汇编指令。每个运算符都是二元的，即有两个参数，第一个参数放在栈顶，第二个参数放在 `ax` 中。这个顺序要特别注意。因为像 `-`，`/` 之类的运算符是与参数顺序有关的。计算后会将栈顶的参数退栈，结果存放在寄存器 `ax` 中。因此计算结束后，两个参数都无法取得了（汇编的意义上，存在内存地址上就另当别论）。

实现如下：

```
else if (op == OR)  ax = *sp++ | ax;
else if (op == XOR) ax = *sp++ ^ ax;
else if (op == AND) ax = *sp++ & ax;
else if (op == EQ)  ax = *sp++ == ax;
else if (op == NE)  ax = *sp++ != ax;
else if (op == LT)  ax = *sp++ < ax;
else if (op == LE)  ax = *sp++ <= ax;
else if (op == GT)  ax = *sp++ >  ax;
else if (op == GE)  ax = *sp++ >= ax;
else if (op == SHL) ax = *sp++ << ax;
else if (op == SHR) ax = *sp++ >> ax;
else if (op == ADD) ax = *sp++ + ax;
else if (op == SUB) ax = *sp++ - ax;
else if (op == MUL) ax = *sp++ * ax;
else if (op == DIV) ax = *sp++ / ax;
else if (op == MOD) ax = *sp++ % ax;
```

内置函数

程序要有用，除了核心的逻辑外还需要输入输出，如 C 语言中我们经常使用的 `printf` 函数就是用于输出。但是 `printf` 函数的实现本身就十分复杂，如果我们的编译器要达到自举，就势必要实现 `printf` 之类的函数，但它又与编译器没有太大的联系，因此我们继续实现新的指令，从虚拟机的角度予以支持。

编译器中我们需要用到的函数有：`exit`, `open`, `close`, `read`, `printf`, `malloc`, `memset` 及 `memcmp`。代码如下：

```
else if (op == EXIT) { printf("exit(%d)", *sp); return *sp;}
else if (op == OPEN) { ax = open((char *)sp[1], sp[0]); }
else if (op == CLOS) { ax = close(*sp);}
else if (op == READ) { ax = read(sp[2], (char *)sp[1], *sp); }
else if (op == PRTF) { tmp = sp + pc[1]; ax = printf((char *)tmp[-1
else if (op == MALC) { ax = (int)malloc(*sp);}
else if (op == MSET) { ax = (int)memset((char *)sp[2], sp[1], *sp);
else if (op == MCMP) { ax = memcmp((char *)sp[2], (char *)sp[1], *s
```

这里的原理是，我们的电脑上已经有了这些函数的实现，因此编译编译器时，这些函数的二进制代码就被编译进了我们的编译器，因此在我们的编译器/虚拟机上运行我们提供的这些指令时，这些函数就是可用的。换句话说就是不需要我们自己去实现了。

最后再加上一个错误判断：

```
else {
printf("unknown instruction:%d\n", op);
return -1;
}
```

测试

下面我们用我们的汇编写一小段程序，来计算 `10+20`，在 `main` 函数中加入下列代码：

```
int main(int argc, char *argv[])
{
ax = 0;
...

i = 0;
text[i++] = IMM;
text[i++] = 10;
text[i++] = PUSH;
text[i++] = IMM;
text[i++] = 20;
text[i++] = ADD;
text[i++] = PUSH;
text[i++] = EXIT;
pc = text;

...
program();
}
```

编译程序 `gcc xc-tutor.c`，运行程序：`./a.out hello.c`。输出

```
exit(30)
```

注意我们的之前的程序需要指令一个源文件，只是现在还用不着，但从结果可以看出，我们的虚拟机还是工作良好的。

小结

本章中我们回顾了计算机的内部运行原理，并仿照 x86 汇编指令设计并实现了我们自己的指令集。

本章的代码可以在 Github 上下载，也可以直接 clone

```
git clone -b step-1 https://github.com/lotabout/write-a-C-interpret
```

实际计算机中，添加一个新的指令需要设计许多新的电路，会增加许多的成本，但我们的需要机中，新的指令几乎不消耗资源，因此我们可以利用这一点，用更多的指令来完成更多的功能，从而简化具体的实现。

手把手教你做一个 C 语言编译器（3）：词法分析器

本章我们要讲解如何构建词法分析器。

本系列：

1. 手把手教你做一个 C 语言编译器（0）：前言
2. 手把手教你做一个 C 语言编译器（1）：设计
3. 手把手教你做一个 C 语言编译器（2）：虚拟机

什么是词法分析器

简而言之，词法分析器用于对源码字符串做预处理，以减少语法分析器的复杂程度。

词法分析器以源码字符串为输入，输出为标记流（token stream），即一连串的标记，每个标记通常包括： `(token, token value)` 即标记本身和标记的值。例如，源码中若包含一个数字 `'998'` ，词法分析器将输出 `(Number, 998)` ，即（数字，998）。再例如：

```
2 + 3 * (4 - 5)
=>
(Number, 2) Add (Number, 3) Multiply Left-Bracket (Number, 4) Subtı
```

通过词法分析器的预处理，语法分析器的复杂度会大大降低，这点在后面的语法分析器我们就能体会。

词法分析器与编译器

要是深入词法分析器，你就会发现，它的本质上也是编译器。我们的编译器是以标记流为输入，输出汇编代码，而词法分析器则是以源码字符串为输入，输出标记流。

```
                   +-------+                      +--------+
-- source code --> | lexer | --> token stream --> | parser | --> as
                   +-------+                      +--------+
```

在这个前提下，我们可以这样认为：直接从源代码编译成汇编代码是很困难的，因为输入的字符串比较难处理。所以我们先编写一个较为简单的编译器（词法分析器）来将字符串转换成标记流，而标记流对于语法分析器而言就容易处理得多了。

词法分析器的实现

由于词法分析的工作很常见，但又枯燥且容易出错，所以人们已经开发出了许多工具来生成词法分析器，如 `lex, flex` 。这些工具允许我们通过正则表达式来识别标记。

这里注意的是，我们并不会一次性地将所有源码全部转换成标记流，原因有二：

1. 字符串转换成标记流有时是有状态的，即与代码的上下文是有关系的。
2. 保存所有的标记流没有意义且浪费空间。

所以实际的处理方法是提供一个函数（即前几篇中提到的 `next()` ），每次调用该函数则返回下一个标记。

支持的标记

在全局中添加如下定义：

```
// tokens and classes (operators last and in precedence order)
enum {
Num = 128, Fun, Sys, Glo, Loc, Id,
Char, Else, Enum, If, Int, Return, Sizeof, While,
Assign, Cond, Lor, Lan, Or, Xor, And, Eq, Ne, Lt, Gt, Le, Ge, Shl,
};
```

这些就是我们要支持的标记符。例如，我们会将 `=` 解析为 `Assign` ；将 `==` 解析为 `Eq` ；将 `!=` 解析为 `Ne` 等等。

所以这里我们会有这样的印象，一个标记（token）可能包含多个字符，且多数情况下如此。而词法分析器能减小语法分析复杂度的原因，正是因为它相当于通过一定的编码（更多的标记）来压缩了源码字符串。

当然，上面这些标记是有顺序的，跟它们在 C 语言中的优先级有关，如 `*(Mul)` 的优先级就要高于 `+(Add)` 。它们的具体使用在后面的语法分析中会提到。

最后要注意的是还有一些字符，它们自己就构成了标记，如右方括号 `]` 或波浪号 `~` 等。我们不另外处理它们的原因是：

1. 它们是单字符的，即并不是多个字符共同构成标记（如 `==` 需要两个字符）；
2. 它们不涉及优先级关系。

词法分析器的框架

即 `next()` 函数的主体：

```
void next() {
char *last_pos;
int hash;

while (token = *src) {
++src;
// parse token here
}
return;
}
```

这里的一个问题是，为什么要用 `while` 循环呢？这就涉及到编译器（记得我们说过词法分析器也是某种意义上的编译器）的一个问题：如何处理错误？

对词法分析器而言，若碰到了一个我们不认识的字符该怎么处理？一般处理的方法有两种：

1. 指出错误发生的位置，并退出整个程序
2. 指出错误发生的位置，跳过当前错误并继续编译

这个 `while` 循环的作用就是跳过这些我们不识别的字符，我们同时还用它来处理空白字符。我们知道，C 语言中空格是用来作为分隔用的，并不作为语法的一部分。因此在实现中我们将它作为“不识别”的字符，这个 `while` 循环可以用来跳过它。

换行符

换行符和空格类似，但有一点不同，每次遇到换行符，我们需要将当前的行号加一：

```
// parse token here
...

if (token == '\n') {
++line;
}
...
```

宏定义

C 语言的宏定义以字符 `#` 开头，如 `# include <stdio.h>` 。我们的编译器并不支持宏定义，所以直接跳过它们。

```
else if (token == '#') {
// skip macro, because we will not support it
while (*src != 0 && *src != '\n') {
src++;
}
}
```

标识符与符号表

标识符（identifier）可以理解为变量名。对于语法分析而言，我们并不关心一个变量具体叫什么名字，而只关心这个变量名代表的唯一标识。例如 `int a;` 定义了变量 `a` ，而之后的语句 `a = 10` ，我们需要知道这两个 `a` 指向的是同一个变量。

基于这个理由，词法分析器会把扫描到的标识符全都保存到一张表中，遇到新的标识符就去查这张表，如果标识符已经存在，就返回它的唯一标识。

那么我们怎么表示标识符呢？如下：

```
struct identifier {
int token;
int hash;
char * name;
int class;
int type;
int value;
int Bclass;
int Btype;
int Bvalue;
}
```

这里解释一下具体的含义：

1. `token` ：该标识符返回的标记，理论上所有的变量返回的标记都应该是 `Id` ，但实际上由于我们还将在符号表中加入关键字如 `if` , `while` 等，它们都有对应的标记。
2. `hash` ：顾名思义，就是这个标识符的哈希值，用于标识符的快速比较。
3. `name` ：存放标识符本身的字符串。
4. `class` ：该标识符的类别，如数字，全局变量或局部变量等。
5. `type` ：标识符的类型，即如果它是个变量，变量是 `int` 型、 `char` 型还是指针型。
6. `value` ：存放这个标识符的值，如标识符是函数，刚存放函数的地址。
7. `BXXXX` ：C 语言中标识符可以是全局的也可以是局部的，当局部标识符的名字与全局标识符相同时，用作保存全局标识符的信息。

由上可以看出，我们实现的词法分析器与传统意义上的词法分析器不太相同。传统意义上的符号表只需要知道标识符的唯一标识即可，而我们还存放了一些只有语法分析器才会得到的信息，如 `type` 。

由于我们的目标是能自举，而我们定义的语法不支持 `struct` ，故而使用下列方式。

```
Symbol table:
----+-----+----+----+----+-----+-----+-----+------+------+----
 .. |token|hash|name|type|class|value|btype|bclass|bvalue| ..
----+-----+----+----+----+-----+-----+-----+------+------+----
    |<---       one single identifier                --->|
```

即用一个整型数组来保存相关的ID信息。每个ID占用数组中的9个空间，分析标识符的相关代码如下：

```
int token_val; // value of current token (mainly for number)
int *current_id, // current parsed ID
*symbols; // symbol table

// fields of identifier
enum {Token, Hash, Name, Type, Class, Value, BType, BClass, BValue,

void next() {
...

else if ((token >= 'a' && token <= 'z') || (token >= 'A' && token <

// parse identifier
last_pos = src - 1;
hash = token;

while ((*src >= 'a' && *src <= 'z') || (*src >= 'A' && *src <= 'Z')
hash = hash * 147 + *src;
src++;
}

// look for existing identifier, linear search
current_id = symbols;
while (current_id[Token]) {
if (current_id[Hash] == hash && !memcmp((char *)current_id[Name], l
//found one, return
token = current_id[Token];
return;
}
current_id = current_id + IdSize;
}

// store new ID
current_id[Name] = (int)last_pos;
current_id[Hash] = hash;
token = current_id[Token] = Id;
return;
}
...
}
```

查找已有标识符的方法是线性查找 `symbols` 表。

数字

数字中较为复杂的一点是需要支持十进制、十六进制及八进制。逻辑也较为直接，可能唯一不好理解的是获取十六进制的值相关的代码。

```
token_val = token_val * 16 + (token & 16) + (token >= 'A' ? 9 : 0);
```

这里要注意的是在ASCII码中，字符 `a` 对应的十六进制值是 `61`，`A` 是 `41`，故通过 `(token & 16)` 可以得到个位数的值。其它就不多说了，这里这样写的目的是装B（其实是抄 c4 的源代码的）。

```
void next() {
...
else if (token >= '0' && token <= '9') {
// parse number, three kinds: dec(123) hex(0x123) oct(017)
token_val = token - '0';
if (token_val > 0) {
// dec, starts with [1-9]
while (*src >= '0' && *src <= '9') {
token_val = token_val*10 + *src++ - '0';
}
} else {
// starts with number 0
if (*src == 'x' || *src == 'X') {
//hex
token = *++src;
while ((token >= '0' && token <= '9') || (token >= 'a' && token <=
token_val = token_val * 16 + (token & 15) + (token >= 'A' ? 9 : 0);
token = *++src;
}
} else {
// oct
while (*src >= '0' && *src <= '7') {
token_val = token_val*8 + *src++ - '0';
}
}
}

token = Num;
return;
}
...
}
```

字符串

在分析时，如果分析到字符串，我们需要将它存放到前一篇文章中说的 `data` 段中。然后返回它在 `data` 段中的地址。另一个特殊的地方是我们需要支持转义符。例如用 `\n` 表示换行符。由于本编译器的目的是达到自己编译自己，所以代码中并没有支持除 `\n` 的转义符，如 `\t`，`\r` 等，但仍支持 `\a` 表示字符 `a` 的语法，如 `\"` 表示 `"`。

在分析时，我们将同时分析单个字符如 `'a'` 和字符串如 `"a string"` 。若得到的是单个字符，我们以 `Num` 的形式返回。相关代码如下：

```
void next() {
...

else if (token == '"' || token == '\'') {
// parse string literal, currently, the only supported escape
// character is '\n', store the string literal into data.
last_pos = data;
while (*src != 0 && *src != token) {
token_val = *src++;
if (token_val == '\\') {
// escape character
token_val = *src++;
if (token_val == 'n') {
token_val = '\n';
}
}

if (token == '"') {
*data++ = token_val;
}
}

src++;
// if it is a single character, return Num token
if (token == '"') {
token_val = (int)last_pos;
} else {
token = Num;
}

return;
}
}
```

注释

在我们的 C 语言中，只支持 `//` 类型的注释，不支持 `/* comments */` 的注释。

```
void next() {
...

else if (token == '/') {
if (*src == '/') {
// skip comments
while (*src != 0 && *src != '\n') {
++src;
}
} else {
// divide operator
token = Div;
return;
}
}

...
}
```

这里我们要额外介绍 `lookahead` 的概念，即提前看多个字符。上述代码中我们看到，除了跳过注释，我们还可能返回除号 `/(Div)` 标记。

提前看字符的原理是：有一个或多个标记是以同样的字符开头的（如本小节中的注释与除号），因此只凭当前的字符我们并无法确定具体应该解释成哪一个标记，所以只能再向前查看字符，如本例需向前查看一个字符，若是 `/` 则说明是注释，反之则是除号。

我们之前说过，词法分析器本质上也是编译器，其实提前看字符的概念也存在于编译器，只是这时就是提前看k个“标记”而不是“字符”了。平时听到的 `LL(k)` 中的 `k` 就是需要向前看的标记的个数了。

另外，我们用词法分析器将源码转换成标记流，能减小语法分析复杂度，原因之一就是减少了语法分析器需要“向前看”的字符个数。

其它

其它的标记的解析就相对容易一些了，我们直接贴上代码：

```
void next() {
...

else if (token == '=') {
// parse '==' and '='
if (*src == '=') {
src ++;
token = Eq;
} else {
token = Assign;
}
```

```
return;
}
else if (token == '+') {
// parse '+' and '++'
if (*src == '+') {
src ++;
token = Inc;
} else {
token = Add;
}
return;
}
else if (token == '-') {
// parse '-' and '--'
if (*src == '-') {
src ++;
token = Dec;
} else {
token = Sub;
}
return;
}
else if (token == '!') {
// parse '!='
if (*src == '=') {
src++;
token = Ne;
}
return;
}
else if (token == '<') {
// parse '<=', '<<' or '<'
if (*src == '=') {
src ++;
token = Le;
} else if (*src == '<') {
src ++;
token = Shl;
} else {
token = Lt;
}
return;
}
else if (token == '>') {
// parse '>=', '>>' or '>'
if (*src == '=') {
src ++;
token = Ge;
} else if (*src == '>') {
src ++;
token = Shr;
} else {
token = Gt;
```

```
}
return;
}
else if (token == '|') {
// parse '|' or '||'
if (*src == '|') {
src ++;
token = Lor;
} else {
token = Or;
}
return;
}
else if (token == '&') {
// parse '&' and '&&'
if (*src == '&') {
src ++;
token = Lan;
} else {
token = And;
}
return;
}
else if (token == '^') {
token = Xor;
return;
}
else if (token == '%') {
token = Mod;
return;
}
else if (token == '*') {
token = Mul;
return;
}
else if (token == '[') {
token = Brak;
return;
}
else if (token == '?') {
token = Cond;
return;
}
else if (token == '~' || token == ';' || token == '{' || token ==
// directly return the character as token;
return;
}

...
}
```

代码较多，但主要逻辑就是向前看一个字符来确定真正的标记。

关键字与内置函数

虽然上面写完了词法分析器，但还有一个问题需要考虑，那就是“关键字”，例如 `if`，`while`，`return` 等。它们不能被作为普通的标识符，因为有特殊的含义。

一般有两种处理方法：

1. 词法分析器中直接解析这些关键字。
2. 在语法分析前将关键字提前加入符号表。

这里我们就采用第二种方法，将它们加入符号表，并提前为它们赋予必要的信息（还记得前面说的标识符 `Token` 字段吗？）。这样当源代码中出现关键字时，它们会被解析成标识符，但由于符号表中已经有了相关的信息，我们就能知道它们是特殊的关键字。

内置函数的行为也和关键字类似，不同的只是赋值的信息，在 `main` 函数中进行初始化如下：

```
// types of variable/function
enum { CHAR, INT, PTR };
int *idmain; // the `main` function

void main() {
...

src = "char else enum if int return sizeof while "
"open read close printf malloc memset memcmp exit void main";

// add keywords to symbol table
i = Char;
while (i <= While) {
next();
current_id[Token] = i++;
}

// add library to symbol table
i = OPEN;
while (i <= EXIT) {
next();
current_id[Class] = Sys;
current_id[Type] = INT;
current_id[Value] = i++;
}

next(); current_id[Token] = Char; // handle void type
next(); idmain = current_id; // keep track of main

...
program();
}
```

代码

本章的代码可以在 Github 上下载，也可以直接 clone

```
git clone -b step-2 https://github.com/lotabout/write-a-C-interpret
```

上面的代码运行后会出现 'Segmentation Falt'，这是正常的，因为它会尝试运行我们上一章创建的虚拟机，但其中并没有任何汇编代码。

小结

本章我们为我们的编译器构建了词法分析器，通过本章的学习，我认为有几个要点需要强调：

1. 词法分析器的作用是对源码字符串进行预处理，作用是减小语法分析器的复杂程度。
2. 词法分析器本身可以认为是一个编译器，输入是源码，输出是标记流。
3. `lookahead(k)` 的概念，即向前看 `k` 个字符或标记。
4. 词法分析中如何处理标识符与符号表。

下一章中，我们将介绍递归下降的语法分析器。我们下一章见。

手把手教你做一个 C 语言编译器（4）：递归下降

本章我们将讲解递归下降的方法，并用它完成一个基本的四则运算的语法分析器。

本系列：

1. 手把手教你做一个 C 语言编译器（0）：前言
2. 手把手教你做一个 C 语言编译器（1）：设计
3. 手把手教你做一个 C 语言编译器（2）：虚拟机
4. 手把手教你做一个 C 语言编译器（3）：词法分析器

什么是递归下降

传统上，编写语法分析器有两种方法，一种是自顶向下，一种是自底自上。自顶向下是从起始非终结符开始，不断地对非终结符进行分解，直到匹配输入的终结符；自底向上是不断地将终结符进行合并，直到合并成起始的非终结符。

其中的自顶向下方法就是我们所说的递归下降。

终结符与非终结符

没有学过编译原理的话可能并不知道什么是“终结符”，“非终结符”。这里我简单介绍一下。首先是 BNF 范式，就是一种用来描述语法的语言，例如，四则运算的规则可以表示如下：

```
<expr> ::= <expr> + <term>
         | <expr> - <term>
         | <term>

<term> ::= <term> * <factor>
         | <term> / <factor>
         | <factor>

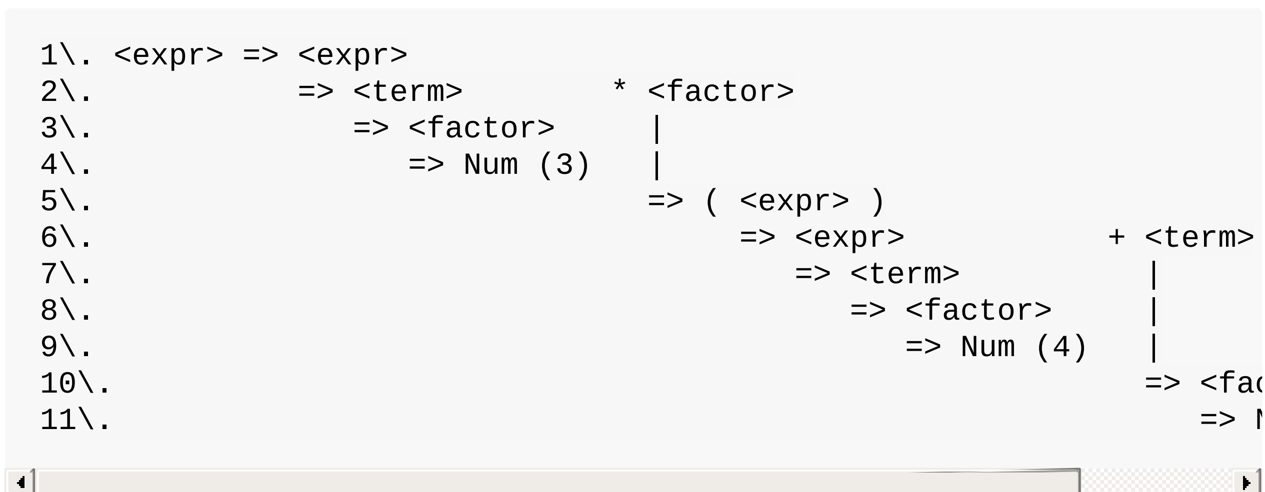
<factor> ::= ( <expr> )
           | Num
```

用尖括号 <> 括起来的就称作 非终结符 ，因为它们可以用 ::= 右侧的式子代替。 | 表示选择，如 <expr> 可以是 <expr> + <term> 、 <expr> - <term> 或 <term> 中的一种。而没有出现在 ::= 左边的就称作 终结符 ，一般终结符对应于词法分析器输出的标记。

四则运算的递归下降

例如，我们对 `3 * (4 + 2)` 进行语法分析。我们假设词法分析器已经正确地将其中的数字识别成了标记 `Num` 。

递归下降是从起始的非终结符开始（顶），本例中是 `<expr>` ，实际中可以自己指定，不指定的话一般认为是第一个出现的非终结符。



```
1\. <expr> => <expr>
2\.           => <term>        * <factor>
3\.              => <factor>     |
4\.                 => Num (3)   |
5\.                              => ( <expr> )
6\.                                   => <expr>           + <term>
7\.                                      => <term>          |
8\.                                         => <factor>     |
9\.                                            => Num (4)   |
10\.                                                    => <fac
11\.                                                       => N
```

可以看到，整个解析的过程是在不断对非终结符进行替换（向下），直到遇见了终结符（底）。而我们可以从解析的过程中看出，一些非终结符如 `<expr>` 被递归地使用了。

为什么选择递归下降

从上小节对四则运算的递归下降解析可以看出，整个解析的过程和语法的 BNF 表示是二分接近的，更为重要的是，我们可以很容易地直接将 BNF 表示转换成实际的代码。方法是为每个产生式（即 `非终结符 ::= ...` ）生成一个同名的函数。

这里会有一个疑问，就是上例中，当一个终结符有多个选择时，如何确定具体选择哪一个？如为什么用 `<expr> ::= <term> * <factor>` 而不是 `<expr> ::= <term> / <factor>` ？这就用到了上一章中提到的“向前看 k 个标记”的概念了。我们向前看一个标记，发现是 `*` ，而这个标记足够让我们确定用哪个表达式了。

另外，递归下下降方法对 BNF 方法本身有一定的要求，否则会有一些问题，如经典的“左递归”问题。

左递归

原则上我们是不讲这么深入，但我们上面的四则运算的文法就是左递归的，而左递归的语法是没法直接使用递归下降的方法实现的。因此我们要消除左递归，消除后的文法如下：

```
<expr> ::= <term> <expr_tail>
<expr_tail> ::= + <term> <expr_tail>
| - <term> <expr_tail>
| <empty>

<term> ::= <factor> <term_tail>
<term_tail> ::= * <factor> <term_tail>
| / <factor> <term_tail>
| <empty>

<factor> ::= ( <expr> )
| Num
```

消除左递归的相关方法，这里不再多说，请自行查阅相关的资料。

四则运算的实现

本节中我们专注语法分析器部分的实现，具体实现很容易，我们直接贴上代码，就是上述的消除左递归后的文法直接转换而来的：

```
int expr();

int factor() {
int value = 0;
if (token == '(') {
match('(');
value = expr();
match(')');
} else {
value = token_val;
match(Num);
}
return value;
}

int term_tail(int lvalue) {
if (token == '*') {
match('*');
int value = lvalue * factor();
return term_tail(value);
} else if (token == '/') {
match('/');
int value = lvalue / factor();
return term_tail(value);
} else {
return lvalue;
}
}
```

```
int term() {
int lvalue = factor();
return term_tail(lvalue);
}

int expr_tail(int lvalue) {
if (token == '+') {
match('+');
int value = lvalue + term();
return expr_tail(value);
} else if (token == '-') {
match('-');
int value = lvalue - term();
return expr_tail(value);
} else {
return lvalue;
}
}

int expr() {
int lvalue = term();
return expr_tail(lvalue);
}
```

可以看到，有了BNF方法后，采用递归向下的方法来实现编译器是很直观的。

我们把词法分析器的代码一并贴上：

```
#include <stdio.h>
#include <stdlib.h>

enum {Num};
int token;
int token_val;
char *line = NULL;
char *src = NULL;

void next() {
// skip white space
while (*src == ' ' || *src == '\t') {
src ++;
}

token = *src++;

if (token >= '0' && token <= '9' ) {
token_val = token - '0';
token = Num;

while (*src >= '0' && *src <= '9') {
token_val = token_val*10 + *src - '0';
src ++;
}
return;
}
}

void match(int tk) {
if (token != tk) {
printf("expected token: %d(%c), got: %d(%c)\n", tk, tk, token, toke
exit(-1);
}
next();
}
```

最后是 `main` 函数：

```
int main(int argc, char *argv[])
{
size_t linecap = 0;
ssize_t linelen;
while ((linelen = getline(&line, &linecap, stdin)) > 0) {
src = line;
next();
printf("%d\n", expr());
}
return 0;
}
```

小结

本章中我们介绍了递归下降的方法，并用它来实现了四则运算的语法分析器。

花这么大精力讲解递归下降方法，是因为几乎所有手工编写的语法分析器都或多或少地有它的影子。换句话说，掌握了递归下降的方法，就可以应付大多数的语法分析器编写。

同时我们也用实例看到了理论（BNF 语法，左递归的消除）是如何帮助我们的工程实现的。尽管理论不是必需的，但如果能掌握它，对于提高我们的水平还是很有帮助的。

手把手教你做一个 C 语言编译器（5）：变量定义

本章中我们用 EBNF 来大致描述我们实现的 C 语言的文法，并实现其中解析变量定义部分。

由于语法分析本身比较复杂，所以我们将它拆分成 3 个部分进行讲解，分别是：变量定义、函数定义、表达式。

本系列：

1. 手把手教你做一个 C 语言编译器（0）：前言
2. 手把手教你做一个 C 语言编译器（1）：设计
3. 手把手教你做一个 C 语言编译器（2）：虚拟机
4. 手把手教你做一个 C 语言编译器（3）：词法分析器
5. 手把手教你做一个 C 语言编译器（4）：递归下降

EBNF 表示

EBNF 是对前一章提到的 BNF 的扩展，它的语法更容易理解，实现起来也更直观。但真正看起来还是很烦，如果不想看可以跳过。

```
program ::= {global_declaration}+

global_declaration ::= enum_decl | variable_decl | function_decl

enum_decl ::= 'enum' [id] '{' id ['=' 'num'] {',' id ['=' 'num'} ']

variable_decl ::= type {'*'} id { ',' {'*'} id } ';'

function_decl ::= type {'*'} id '(' parameter_decl ')' '{' body_dec

parameter_decl ::= type {'*'} id {',' type {'*'} id}

body_decl ::= {variable_decl}, {statement}

statement ::= non_empty_statement | empty_statement

non_empty_statement ::= if_statement | while_statement | '{' stater
| 'return' expression | expression ';'

if_statement ::= 'if' '(' expression ')' statement ['else' non_empt

while_statement ::= 'while' '(' expression ')' non_empty_statement
```

其中 `expression` 相关的内容我们放到后面解释，主要原因是我们的语言不支持跨函数递归，而为了实现自举，实际上我们也不能使用递归（亏我们说了一章的递归下降）。

P.S. 我是先写程序再总结上面的文法，所以实际上它们间的对应关系并不是特别明显。

解析变量的定义

本章要讲解的就是上节文法中的 `enum_decl` 和 `variable_decl` 部分。

program()

首先是之前定义过的 `program` 函数，将它改成：

```
void program() {
// get next token
next();
while (token > 0) {
global_declaration();
}
}
```

我知道 `global_declaration` 函数还没有出现过，但没有关系，采用自顶向下的编写方法就是要不断地实现我们需要的内容。下面是 `global_declaration` 函数的内容：

global_declaration()

即全局的定义语句，包括变量定义，类型定义（只支持枚举）及函数定义。代码如下：

```
int basetype; // the type of a declaration, make it global for conv
int expr_type; // the type of an expression

void global_declaration() {
// global_declaration ::= enum_decl | variable_decl | function_decl
//
// enum_decl ::= 'enum' [id] '{' id ['=' 'num'] {',' id ['=' 'num']
//
// variable_decl ::= type {'*'} id { ',' {'*'} id } ';'
//
// function_decl ::= type {'*'} id '(' parameter_decl ')' '{' body_

int type; // tmp, actual type for variable
int i; // tmp
```

```
basetype = INT;

// parse enum, this should be treated alone.
if (token == Enum) {
// enum [id] { a = 10, b = 20, ... }
match(Enum);
if (token != '{') {
match(Id); // skip the [id] part
}
if (token == '{') {
// parse the assign part
match('{');
enum_declaration();
match('}');
}

match(';');
return;
}

// parse type information
if (token == Int) {
match(Int);
}
else if (token == Char) {
match(Char);
basetype = CHAR;
}

// parse the comma seperated variable declaration.
while (token != ';' && token != '}') {
type = basetype;
// parse pointer type, note that there may exist `int ****x;`
while (token == Mul) {
match(Mul);
type = type + PTR;
}

if (token != Id) {
// invalid declaration
printf("%d: bad global declaration\n", line);
exit(-1);
}
if (current_id[Class]) {
// identifier exists
printf("%d: duplicate global declaration\n", line);
exit(-1);
}
match(Id);
current_id[Type] = type;

if (token == '(') {
```

```
current_id[Class] = Fun;
current_id[Value] = (int)(text + 1); // the memory address of funct
function_declaration();
} else {
// variable declaration
current_id[Class] = Glo; // global variable
current_id[Value] = (int)data; // assign memory address
data = data + sizeof(int);
}

if (token == ',') {
match(',');
}
}
next();
}
```

看了上面的代码，能大概理解吗？这里我们讲解其中的一些细节。

向前看标记 ：其中的 `if (token == xxx)` 语句就是用来向前查看标记以确定使用哪一个产生式，例如只要遇到 `enum` 我们就知道是需要解析枚举类型。而如果只解析到类型，如 `int identifier` 时我们并不能确定 `identifier` 是一个普通的变量还是一个函数，所以还需要继续查看后续的标记，如果遇到 `(` 则可以断定是函数了，反之则是变量。

变量类型的表示 ：我们的编译器支持指针类型，那意味着也支持指针的指针，如 `int **data;` 。那么我们如何表示指针类型呢？前文中我们定义了支持的类型：

```
// types of variable/function
enum { CHAR, INT, PTR };
```

所以一个类型首先有基本类型，如 `CHAR` 或 `INT` ，当它是一个指向基本类型的指针时，如 `int *data` ，我们就将它的类型加上 `PTR` 即代码中的： `type = type + PTR;` 。同理，如果是指针的指针，则再加上 `PTR` 。

enum_declaration()

用于解析枚举类型的定义。主要的逻辑用于解析用逗号（ `,` ）分隔的变量，值得注意的是在编译器中如何保存枚举变量的信息。

即我们将该变量的类别设置成了 `Num` ，这样它就成了全局的常量了，而注意到上节中，正常的全局变量的类别则是 `Glo` ，类别信息在后面章节中解析 `expression` 会使用到。

```
void enum_declaration() {
// parse enum [id] { a = 1, b = 3, ...}
int i;
i = 0;
while (token != '}') {
if (token != Id) {
printf("%d: bad enum identifier %d\n", line, token);
exit(-1);
}
next();
if (token == Assign) {
// like {a=10}
next();
if (token != Num) {
printf("%d: bad enum initializer\n", line);
exit(-1);
}
i = token_val;
next();
}

current_id[Class] = Num;
current_id[Type] = INT;
current_id[Value] = i++;

if (token == ',') {
next();
}
}
}
```

其它

其中的 `function_declaration` 函数我们将放到下一章中讲解。`match` 函数是一个辅助函数：

```
void match(int tk) {
if (token == tk) {
next();
} else {
printf("%d: expected token: %d\n", line, tk);
exit(-1);
}
}
```

它将 `next` 函数包装起来，如果不是预期的标记则报错并退出。

代码

本章的代码可以在 Github 上下载，也可以直接 clone

```
git clone -b step-3 https://github.com/lotabout/write-a-C-interpret
```

本章的代码还无法正常运行，因为还有许多功能没有实现，但如果有兴趣的话，可以自己先试着去实现它。

小结

本章的内容应该不难，除了开头的 EBNF 表达式可能相对不好理解一些，但如果你查看了 EBNF 的具体表示方法后就不难理解了。

剩下的内容就是按部就班地将 EBNF 的产生式转换成函数的过程，如果你理解了上一章中的内容，相信这部分也不难理解。

下一章中我们将介绍如何解析函数的定义，敬请期待。

手把手教你做一个 C 语言编译器（6）：函数定义

由于语法分析本身比较复杂，所以我们将它拆分成 3 个部分进行讲解，分别是：变量定义、函数定义、表达式。本章讲解函数定义相关的内容。

本系列：

EBNF 表示

这是上一章的 EBNF 方法中与函数定义相关的内容。

```
variable_decl ::= type {'*'} id { ',' {'*'} id } ';'

function_decl ::= type {'*'} id '(' parameter_decl ')' '{' body_dec

parameter_decl ::= type {'*'} id {',' type {'*'} id}

body_decl ::= {variable_decl}, {statement}

statement ::= non_empty_statement | empty_statement

non_empty_statement ::= if_statement | while_statement | '{' stater
| 'return' expression | expression ';'

if_statement ::= 'if' '(' expression ')' statement ['else' non_empt

while_statement ::= 'while' '(' expression ')' non_empty_statement
```

解析函数的定义

上一章的代码中，我们已经知道了什么时候开始解析函数的定义，相关的代码如下：

```
...
if (token == '(') {
current_id[Class] = Fun;
current_id[Value] = (int)(text + 1); // the memory address of funct
function_declaration();
} else {
...
```

即在这断代码之前，我们已经为当前的标识符（identifier）设置了正确的类型，上面这断代码为当前的标识符设置了正确的类别（Fun），以及该函数在代码段（text segment）中的位置。接下来开始解析函数定义相关的内容：`parameter_decl` 及 `body_decl` 。

函数参数与汇编代码

现在我们要回忆如何将“函数”转换成对应的汇编代码，因为这决定了在解析时我们需要哪些相关的信息。考虑下列函数：

```
int demo(int param_a, int *param_b) {
int local_1;
char local_2;

...
}
```

那么它应该被转换成什么样的汇编代码呢？在思考这个问题之前，我们需要了解当 `demo` 函数被调用时，计算机的栈的状态，如下（参照第三章讲解的虚拟机）：

```
|    ....       | high address
+---------------+
| arg: param_a  |    new_bp + 3
+---------------+
| arg: param_b  |    new_bp + 2
+---------------+
|return address |    new_bp + 1
+---------------+
| old BP        | <- new BP
+---------------+
| local_1       |    new_bp - 1
+---------------+
| local_2       |    new_bp - 2
+---------------+
|    ....       |  low address
```

这里最为重要的一点是，无论是函数的参数（如 `param_a` ）还是函数的局部变量（如 `local_1` ）都是存放在计算机的 栈 上的。因此，与存放在 数据段 中的全局变量不同，在函数内访问它们是通过 `new_bp` 指针和对应的位移量进行的。因此，在解析的过程中，我们需要知道参数的个数，各个参数的位移量。

函数定义的解析

这相当于是整个函数定义的语法解析的框架，代码如下：

```
void function_declaration() {
// type func_name (...) {...}
//               | this part

match('(');
function_parameter();
match(')');
match('{');
function_body();
//match('}');                 //  ①

// ②
// unwind local variable declarations for all local variables.
current_id = symbols;
while (current_id[Token]) {
if (current_id[Class] == Loc) {
current_id[Class] = current_id[BClass];
current_id[Type]  = current_id[BType];
current_id[Value] = current_id[BValue];
}
current_id = current_id + IdSize;
}
}
```

其中①中我们没有消耗最后的 `}` 字符。这么做的原因是： `variable_decl` 与 `function_decl` 是放在一起解析的，而 `variable_decl` 是以字符 `;` 结束的。而 `function_decl` 是以字符 `}` 结束的，若在此通过 `match` 消耗了 ‘;’ 字符，那么外层的 `while` 循环就没法准确地知道函数定义已经结束。所以我们将结束符的解析放在了外层的 `while` 循环中。

而②中的代码是用于将符号表中的信息恢复成全局的信息。这是因为，局部变量是可以和全局变量同名的，一旦同名，在函数体内局部变量就会覆盖全局变量，出了函数体，全局变量就恢复了原先的作用。这段代码线性地遍历所有标识符，并将保存在 `BXXX` 中的信息还原。

解析参数

```
parameter_decl ::= type {'*'} id {',' type {'*'} id}
```

解析函数的参数就是解析以逗号分隔的一个个标识符，同时记录它们的位置与类型。

```
int index_of_bp; // index of bp pointer on stack

void function_parameter() {
int type;
int params;
params = 0;
while (token != ')') {
// ①

// int name, ...
type = INT;
if (token == Int) {
match(Int);
} else if (token == Char) {
type = CHAR;
match(Char);
}

// pointer type
while (token == Mul) {
match(Mul);
type = type + PTR;
}

// parameter name
if (token != Id) {
printf("%d: bad parameter declaration\n", line);
exit(-1);
}
if (current_id[Class] == Loc) {
printf("%d: duplicate parameter declaration\n", line);
exit(-1);
}

match(Id);

//②
// store the local variable
current_id[BClass] = current_id[Class]; current_id[Class]  = Loc;
current_id[BType]  = current_id[Type];  current_id[Type]   = type;
current_id[BValue] = current_id[Value]; current_id[Value]  = params
if (token == ',') {
match(',');
}
}

// ③
index_of_bp = params+1;
}
```

其中①与全局变量定义的解析十分一样，用于解析该参数的类型。

而②则与上节中提到的“局部变量覆盖全局变量”相关，先将全局变量的信息保存（无论是是否真的在全局中用到了这个变量）在 `BXXX` 中，再赋上局部变量相关的信息，如 `Value` 中存放的是参数的位置（是第几个参数）。

③则与汇编代码的生成有关，`index_of_bp` 就是前文提到的 `new_bp` 的位置。

函数体的解析

我们实现的 C 语言与现代的 C 语言不太一致，我们需要所有的变量定义出现在所有的语句之前。函数体的代码如下：

```
void function_body() {
// type func_name (...) {...}
//                   -->|   |<--

// ... {
// 1\. local declarations
// 2\. statements
// }

int pos_local; // position of local variables on the stack.
int type;
pos_local = index_of_bp;

// ①
while (token == Int || token == Char) {
// local variable declaration, just like global ones.
basetype = (token == Int) ? INT : CHAR;
match(token);

while (token != ';') {
type = basetype;
while (token == Mul) {
match(Mul);
type = type + PTR;
}

if (token != Id) {
// invalid declaration
printf("%d: bad local declaration\n", line);
exit(-1);
}
if (current_id[Class]) {
// identifier exists
printf("%d: duplicate local declaration\n", line);
exit(-1);
}
match(Id);
```

```
// store the local variable
current_id[BClass] = current_id[Class]; current_id[Class]  = Loc;
current_id[BType]  = current_id[Type];  current_id[Type]   = type;
current_id[BValue] = current_id[Value]; current_id[Value]  = ++pos_

if (token == ',') {
match(',');
}
}
match(';');
}

// ②
// save the stack size for local variables
*++text = ENT;
*++text = pos_local - index_of_bp;

// statements
while (token != '}') {
statement();
}

// emit code for leaving the sub function
*++text = LEV;
}
```

其中①用于解析函数体内的局部变量的定义，代码与全局的变量定义几乎一样。

而②则用于生成汇编代码，我们在第三章的虚拟机中提到过，我们需要在栈上为局部变量预留空间，这两行代码起的就是这个作用。

代码

本章的代码可以在 Github 上下载，也可以直接 clone

```
git clone -b step-4 https://github.com/lotabout/write-a-C-interpret
```

本章的代码依旧无法运行，还有两个重要函数没有完成： `statement` 及 `expression` ，感兴趣的话可以尝试自己实现它们。

小结

本章中我们用了不多的代码完成了函数定义的解析。大部分的代码依旧是用于解析变量：参数和局部变量，而它们的逻辑和全局变量的解析几乎一致，最大的区别就是保存的信息不同。

当然，要理解函数定义的解析过程，最重要的是理解我们会为函数生成怎样的汇编代码，因为这决定了我们需要从解析中获取什么样的信息（例如参数的位置，个数等），而这些可能需要你重新回顾一下“虚拟机”这一章，或是重新学习学习汇编相关的知识。

下一章中我们将讲解最复杂的表达式的解析，同时也是整个编译器最后的部分，敬请期待。

手把手教你做一个 C 语言编译器（7）：语句

整个编译器还剩下最后两个部分：语句和表达式的解析。它们的内容比较多，主要涉及如何将语句和表达式编译成汇编代码。这章讲解语句的解析，相对于表达式来说它还是较为容易的。

本系列：

1. 手把手教你做一个 C 语言编译器（0）：前言
2. 手把手教你做一个 C 语言编译器（1）：设计
3. 手把手教你做一个 C 语言编译器（2）：虚拟机
4. 手把手教你做一个 C 语言编译器（3）：词法分析器
5. 手把手教你做一个 C 语言编译器（4）：递归下降
6. 手把手教你做一个 C 语言编译器（5）：变量定义
7. 手把手教你做一个 C 语言编译器（6）：函数定义

语句

C 语言区分“语句”（statement）和“表达式”（expression）两个概念。简单地说，可以认为语句就是表达式加上末尾的分号。

在我们的编译器中共识别 6 种语句：

1. `if (...) <statement> [else <statement>]`
2. `while (...) <statement>`
3. `{ <statement> }`
4. `return xxx;`
5. `<empty statement>;`
6. `expression;` (expression end with semicolon)

它们的语法分析都相对容易，重要的是去理解如何将这些语句编译成汇编代码，下面我们逐一解释。

IF 语句

IF 语句的作用是跳转，跟据条件表达式决定跳转的位置。我们看看下面的伪代码：

```
if (...) <statement> [else <statement>]

if (<cond>)                   <cond>
JZ a
<true_statement>   ===>     <true_statement>
else:                         JMP b
a:                           a:
<false_statement>           <false_statement>
b:                           b:
```

对应的汇编代码流程为：

1. 执行条件表达式 `<cond>`。
2. 如果条件失败，则跳转到 `a` 的位置，执行 `else` 语句。这里 `else` 语句是可以省略的，此时 `a` 和 `b` 都指向 IF 语句后方的代码。
3. 因为汇编代码是顺序排列的，所以如果执行了 `true_statement`，为了防止因为顺序排列而执行了 `false_statement`，所以需要无条件跳转 `JMP b`。

对应的 C 代码如下：

```
if (token == If) {
match(If);
match('(');
expression(Assign); // parse condition
match(')');

*++text = JZ;
b = ++text;

statement(); // parse statement
if (token == Else) { // parse else
match(Else);

// emit code for JMP B
*b = (int)(text + 3);
*++text = JMP;
b = ++text;

statement();
}

*b = (int)(text + 1);
}
```

While 语句

While 语句比 If 语句简单，它对应的汇编代码如下：

```
a:                       a:
while (<cond>)          <cond>
JZ b
<statement>            <statement>
JMP a
b:                       b:
```

没有什么值得说明的内容，它的 C 代码如下：

```
else if (token == While) {
match(While);

a = text + 1;

match('(');
expression(Assign);
match(')');

*++text = JZ;
b = ++text;

statement();

*++text = JMP;
*++text = (int)a;
*b = (int)(text + 1);
}
```

Return 语句

Return 唯一特殊的地方是：一旦遇到了 Return 语句，则意味着函数要退出了，所以需要生成汇编代码 `LEV` 来表示退出。

```
else if (token == Return) {
// return [expression];
match(Return);

if (token != ';') {
expression(Assign);
}

match(';');

// emit code for return
*++text = LEV;
}
```

其它语句

其它语句并不直接生成汇编代码，所以不多做说明，代码如下：

```
else if (token == '{') {
// { <statement> ... }
match('{');

while (token != '}') {
statement();
}

match('}');
}
else if (token == ';') {
// empty statement
match(';');
}
else {
// a = b; or function_call();
expression(Assign);
match(';');
}
```

代码

本章的代码可以在 Github 上下载，也可以直接 clone

```
git clone -b step-5 https://github.com/lotabout/write-a-C-interpret
```

本章的代码依旧无法运行，还剩最后一部分没有完成： `expression` 。

小结

本章讲解了如何将语句编译成汇编代码，内容相对容易一些，关键就是去理解汇编代码的执行原理。

同时值得一提的是，编译器的语法分析部分其实是很简单的，而真正的难点是如何在语法分析时收集足够多的信息，最终把源代码转换成目标代码（汇编）。我认为这也是初学者实现编译器的一大难点，往往比词法分析/语法分析更困难。

所以建议如果没有学过汇编，可以学习学习，它本身不难，但对理解计算机的原理有很大帮助。

手把手教你做一个 C 语言编译器（8）：表达式

这是整个编译器的最后一部分，解析表达式。什么是表达式？表达式是将各种语言要素的一个组合，用来求值。例如：函数调用、变量赋值、运算符运算等等。

表达式的解析难点有二：一是运算符的优先级问题，二是如何将表达式编译成目标代码。我们就来逐一说明。

本系列：

1. 手把手教你做一个 C 语言编译器（0）：前言
2. 手把手教你做一个 C 语言编译器（1）：设计
3. 手把手教你做一个 C 语言编译器（2）：虚拟机
4. 手把手教你做一个 C 语言编译器（3）：词法分析器
5. 手把手教你做一个 C 语言编译器（4）：递归下降
6. 手把手教你做一个 C 语言编译器（5）：变量定义
7. 手把手教你做一个 C 语言编译器（6）：函数定义
8. 手把手教你做一个 C 语言编译器（7）：语句

运算符的优先级

运算符的优先级决定了表达式的运算顺序，如在普通的四则运算中，乘法 `*` 优先级高于加法 `+`，这就意味着表达式 `2 + 3 * 4` 的实际运行顺序是 `2 + (3 * 4)` 而不是 `(2 + 3) * 4`。

C 语言定义了各种表达式的优先级，可以参考 C 语言运算符优先级。

传统的编程书籍会用“逆波兰式”实现四则运算来讲解优先级问题。实际上，优先级关心的就是哪个运算符先计算，哪个运算符后计算（毕竟叫做“优先级”嘛）。而这就意味着我们需要决定先为哪个运算符生成目标代码（汇编），因为汇编代码是顺序排列的，我们必须先计算优先级高的运算符。

那么如何确定运算符的优先级呢？答曰：栈（递归调用的实质也是栈的处理）。

举一个例子：`2 + 3 - 4 * 5`，它的运算顺序是这样的：

1. 将 `2` 入栈
2. 遇到运算符 `+`，入栈，此时我们期待的是 `+` 的另一个参数
3. 遇到数字 `3`，原则上我们需要立即计算 `2+3` 的值，但我们不确定数字 `3` 是否属于优先级更高的运算符，所以先将它入栈。
4. 遇到运算符 `-`，它的优先级和 `+` 相同，此时判断参数 `3` 属于这前的 `+`。将运算符 `+` 出栈，并将之前的 `2` 和 `3` 出栈，计算 `2+3` 的结果，得到 `5` 入栈。同时将运算符 `-` 入栈。
5. 遇到数字 `4`，同样不能确定是否能立即计算，入栈
6. 遇到运算符 `*` 优先级大于 `-`，入栈
7. 遇到数字 `5`，依旧不能确定是否立即计算，入栈

8. 表达式结束，运算符出栈，为 `*`，将参数出栈，计算 `4*5` 得到结果 `20` 入栈。
9. 运算符出栈，为 `-`，将参数出栈，计算 `5-20`，得到 `-15` 入栈。
10. 此时运算符栈为空，因此得到结果 `-15`。

```
// after step 1, 2
|      |
+------+
| 3    |   |      |
+------+   +------+
| 2    |   | +    |
+------+   +------+

// after step 4
|      |   |      |
+------+   +------+
| 5    |   | -    |
+------+   +------+

// after step 7
|      |
+------+
| 5    |
+------+   +------+
| 4    |   | *    |
+------+   +------+
| 5    |   | -    |
+------+   +------+
```

综上，在计算一个运算符‘x’之前，必须先查看它的右方，找出并计算所有优先级大于‘x’的运算符，之后再计算运算符‘x’。

最后注意的是优先通常只与多元运算符相关，单元运算符往往没有这个问题（因为只有一个参数）。也可以认为“优先级”的实质就是两个运算符在抢参数。

一元运算符

上节中说到了运算符的优先级，也提到了优先级一般只与多元运算符有关，这也意味着一元运算符的优先级总是高于多元运算符。因为我们需要先对它们进行解析。

当然，这部分也将同时解析参数本身（如变量、数字、字符串等等）。

关于表达式的解析，与语法分析相关的部分就是上文所说的优先级问题了，而剩下的较难较烦的部分是与目标代码的生成有关的。因此对于需要讲解的运算符，我们主要从它的目标代码入手。

常量

首先是数字，用 `IMM` 指令将它加载到 `AX` 中即可：

```
if (token == Num) {
match(Num);

// emit code
*++text = IMM;
*++text = token_val;
expr_type = INT;
}
```

接着是字符串常量。它比较特殊的一点是 C 语言的字符串常量支持如下风格：

```
char *p;
p = "first line"
"second line";
```

即跨行的字符串拼接，它相当于：

```
char *p;
p = "first linesecond line";
```

所以解析的时候要注意这一点：

```
else if (token == '"') {
// emit code
*++text = IMM;
*++text = token_val;

match('"');
// store the rest strings
while (token == '"') {
match('"');
}

// append the end of string character '', all the data are default
// to 0, so just move data one position forward.
data = (char *)(((int)data + sizeof(int)) & (-sizeof(int)));
expr_type = PTR;
}
```

sizeof

`sizeof` 是一个一元运算符，我们需要知道后面参数的类型，类型的解析在前面的文章中我们已经很熟悉了。

```
else if (token == Sizeof) {
// sizeof is actually an unary operator
// now only `sizeof(int)`, `sizeof(char)` and `sizeof(*...)` are
// supported.
match(Sizeof);
match('(');
expr_type = INT;

if (token == Int) {
match(Int);
} else if (token == Char) {
match(Char);
expr_type = CHAR;
}

while (token == Mul) {
match(Mul);
expr_type = expr_type + PTR;
}

match(')');

// emit code
*++text = IMM;
*++text = (expr_type == CHAR) ? sizeof(char) : sizeof(int);

expr_type = INT;
}
```

注意的是只支持 `sizeof(int)` ，`sizeof(char)` 及 `sizeof(pointer type...)` 。并且它的结果是 `int` 型。

变量与函数调用

由于取变量的值与函数的调用都是以 `Id` 标记开头的，因此将它们放在一起处理。

```
else if (token == Id) {
// there are several type when occurs to Id
// but this is unit, so it can only be
// 1\. function call
// 2\. Enum variable
// 3\. global/local variable
match(Id);

id = current_id;

if (token == '(') {
// function call
```

```
match('(');

// ①
// pass in arguments
tmp = 0; // number of arguments
while (token != ')') {
expression(Assign);
*++text = PUSH;
tmp ++;

if (token == ',') {
match(',');
}
}
match(')');

// ②
// emit code
if (id[Class] == Sys) {
// system functions
*++text = id[Value];
}
else if (id[Class] == Fun) {
// function call
*++text = CALL;
*++text = id[Value];
}
else {
printf("%d: bad function call\n", line);
exit(-1);
}

// ③
// clean the stack for arguments
if (tmp > 0) {
*++text = ADJ;
*++text = tmp;
}
expr_type = id[Type];
}
else if (id[Class] == Num) {
// ④
// enum variable
*++text = IMM;
*++text = id[Value];
expr_type = INT;
}
else {
// ⑤
// variable
if (id[Class] == Loc) {
*++text = LEA;
*++text = index_of_bp - id[Value];
```

```
}
else if (id[Class] == Glo) {
*++text = IMM;
*++text = id[Value];
}
else {
printf("%d: undefined variable\n", line);
exit(-1);
}

//⑥
// emit code, default behaviour is to load the value of the
// address which is stored in `ax`
expr_type = id[Type];
*++text = (expr_type == Char) ? LC : LI;
}
}
```

①中注意我们是顺序将参数入栈，这和第三章：虚拟机中讲解的指令是对应的。与之不同，标准 C 是逆序将参数入栈的。

②中判断函数的类型，同样在第三章：“虚拟机”中我们介绍过内置函数的支持，如 `printf` , `read` , `malloc` 等等。内置函数有对应的汇编指令，而普通的函数则编译成 `CALL <addr>` 的形式。

③用于清除入栈的参数。因为我们不在乎出栈的值，所以直接修改栈指针的大小即可。

④：当该标识符是全局定义的枚举类型时，直接将对应的值用 `IMM` 指令存入 `AX` 即可。

⑤则是用于加载变量的值，如果是局部变量则采用与 `bp` 指针相对位置的形式（参见第 7章函数定义）。而如果是全局变量则用 `IMM` 加载变量的地址。

⑥：无论是全局还是局部变量，最终都根据它们的类型用 `LC` 或 `LI` 指令加载对应的值。

关于变量，你可能有疑问，如果遇到标识符就用 `LC/LI` 载入相应的值，那诸如 `a[10]` 之类的表达式要如何实现呢？后面我们会看到，根据标识符后的运算符，我们可能会修改或删除现有的 `LC/LI` 指令。

强制转换

虽然我们前面没有提到，但我们一直用 `expr_type` 来保存一个表达式的类型，强制转换的作用是获取转换的类型，并直接修改 `expr_type` 的值。

```
else if (token == '(') {
// cast or parenthesis
match('(');
if (token == Int || token == Char) {
tmp = (token == Char) ? CHAR : INT; // cast type
match(token);
while (token == Mul) {
match(Mul);
tmp = tmp + PTR;
}

match(')');

expression(Inc); // cast has precedence as Inc(++)

expr_type  = tmp;
} else {
// normal parenthesis
expression(Assign);
match(')');
}
}
```

指针取值

诸如 `*a` 的指针取值，关键是判断 `a` 的类型，而就像上节中提到的，当一个表达式解析结束时，它的类型保存在变量 `expr_type` 中。

```
else if (token == Mul) {
// dereference *<addr>
match(Mul);
expression(Inc); // dereference has the same precedence as Inc(++)

if (expr_type >= PTR) {
expr_type = expr_type - PTR;
} else {
printf("%d: bad dereference\n", line);
exit(-1);
}

*++text = (expr_type == CHAR) ? LC : LI;
}
```

取址操作

这里我们就能看到“变量与函数调用”一节中所说的修改或删除 `LC/LI` 指令了。前文中我们说到，对于变量，我们会先加载它的地址，并根据它们类型使用 `LC/LI` 指令加载实际内容，例如对变量 `a`：

```
IMM <addr>
LI
```

那么对变量 `a` 取址，其实只要不执行 `LC/LI` 即可。因此我们删除相应的指令。

```
else if (token == And) {
// get the address of
match(And);
expression(Inc); // get the address of
if (*text == LC || *text == LI) {
text --;
} else {
printf("%d: bad address of\n", line);
exit(-1);
}

expr_type = expr_type + PTR;
}
```

逻辑取反

我们没有直接的逻辑取反指令，因此我们判断它是否与数字 0 相等。而数字 0 代表了逻辑 “False”。

```
else if (token == '!') {
// not
match('!');
expression(Inc);

// emit code, use <expr> == 0
*++text = PUSH;
*++text = IMM;
*++text = 0;
*++text = EQ;

expr_type = INT;
}
```

按位取反

同样我们没有相应的指令，所以我们用异或来实现，即 `~a = a ^ 0xFFFF`。

```
else if (token == '~') {
// bitwise not
match('~');
expression(Inc);

// emit code, use <expr> XOR -1
*++text = PUSH;
*++text = IMM;
*++text = -1;
*++text = XOR;

expr_type = INT;
}
```

正负号

注意这里并不是四则运算中的加减法，而是单个数字的取正取负操作。同样，我们没有取负的操作，用 `0 - x` 来实现 `-x` 。

```
else if (token == Add) {
// +var, do nothing
match(Add);
expression(Inc);

expr_type = INT;
}
else if (token == Sub) {
// -var
match(Sub);

if (token == Num) {
*++text = IMM;
*++text = -token_val;
match(Num);
} else {

*++text = IMM;
*++text = -1;
*++text = PUSH;
expression(Inc);
*++text = MUL;
}

expr_type = INT;
}
```

自增自减

注意的是自增自减操作的优先级是和它的位置有关的。如 `++p` 的优先级高于 `p++`，这里我们解析的就是类似 `++p` 的操作。

```
else if (token == Inc || token == Dec) {
tmp = token;
match(token);
expression(Inc);
// ①
if (*text == LC) {
*text = PUSH; // to duplicate the address
*++text = LC;
} else if (*text == LI) {
*text = PUSH;
*++text = LI;
} else {
printf("%d: bad lvalue of pre-increment\n", line);
exit(-1);
}
*++text = PUSH;
*++text = IMM;
// ②
*++text = (expr_type > PTR) ? sizeof(int) : sizeof(char);
*++text = (tmp == Inc) ? ADD : SUB;
*++text = (expr_type == CHAR) ? SC : SI;
}
```

对应的汇编代码也比较直观，只是在实现 `++p` 时，我们要使用变量 `p` 的地址两次，所以我们需要先 `PUSH` （①）。

②则是因为自增自减操作还需要处理是指针的情形。

二元运算符

这里，我们需要处理多运算符的优先级问题，就如前文的“优先级”一节提到的，我们需要不断地向右扫描，直到遇到优先级 小于 当前优先级的运算符。

回想起我们之前定义过的各个标记，它们是以优先级从低到高排列的，即 `Assign` 的优先级最低，而 `Brak` （ `[` ） 的优先级最高。

```
enum {
Num = 128, Fun, Sys, Glo, Loc, Id,
Char, Else, Enum, If, Int, Return, Sizeof, While,
Assign, Cond, Lor, Lan, Or, Xor, And, Eq, Ne, Lt, Gt, Le, Ge, Shl,
};
```

所以，当我们调用 `expression(level)` 进行解析的时候，我们其实通过了参数 `level` 指定了当前的优先级。在前文的一元运算符处理中也用到了这一点。

所以，此时的二元运算符的解析的框架为：

```
while (token >= level) {
// parse token for binary operator and postfix operator
}
```

解决了优先级的问题，让我们继续讲解如何把运算符编译成汇编代码吧。

赋值操作

赋值操作是优先级最低的运算符。考虑诸如 `a = (expession)` 的表达式，在解析 `=` 之前，我们已经为变量 `a` 生成了如下的汇编代码：

```
IMM <addr>
LC/LI
```

当解析完 `=` 右边的表达式后，相应的值会存放在 `ax` 中，此时，为了实际将这个值保存起来，我们需要类似下面的汇编代码：

```
IMM <addr>
PUSH
SC/SI
```

明白了这点，也就能理解下面的源代码了：

```
tmp = expr_type;
if (token == Assign) {
// var = expr;
match(Assign);
if (*text == LC || *text == LI) {
*text = PUSH; // save the lvalue's pointer
} else {
printf("%d: bad lvalue in assignment\n", line);
exit(-1);
}
expression(Assign);

expr_type = tmp;
*++text = (expr_type == CHAR) ? SC : SI;
}
```

三目运算符

这是 C 语言中唯一的一个三元运算符： `? :` ，它相当于一个小型的 If 语句，所以生成的代码也类似于 If 语句，这里就不多作解释。

```
else if (token == Cond) {
// expr ? a : b;
match(Cond);
*++text = JZ;
addr = ++text;
expression(Assign);
if (token == ':') {
match(':');
} else {
printf("%d: missing colon in conditional\n", line);
exit(-1);
}
*addr = (int)(text + 3);
*++text = JMP;
addr = ++text;
expression(Cond);
*addr = (int)(text + 1);
}
```

逻辑运算符

这包括 `||` 和 `&&` 。它们对应的汇编代码如下：

```
<expr1> || <expr2>     <expr1> && <expr2>

...<expr1>...          ...<expr1>...
JNZ b                  JZ b
...<expr2>...          ...<expr2>...
b:                     b:
```

所以源码如下：

```
else if (token == Lor) {
// logic or
match(Lor);
*++text = JNZ;
addr = ++text;
expression(Lan);
*addr = (int)(text + 1);
expr_type = INT;
}
else if (token == Lan) {
// logic and
match(Lan);
*++text = JZ;
addr = ++text;
expression(Or);
*addr = (int)(text + 1);
expr_type = INT;
}
```

数学运算符

它们包括 `|` , `^` , `&` , `==` , `!=` `<=` , `>=` , `<` , `>` , `<<` , `>>` , `+` , `-` , `*` , `/` , `%` 。它们的实现都很类似，我们以异或 `^` 为例：

```
<expr1> ^ <expr2>

...<expr1>...          <- now the result is on ax
PUSH
...<expr2>...          <- now the value of <expr2> is on ax
XOR
```

所以它对应的代码为：

```
else if (token == Xor) {
// bitwise xor
match(Xor);
*++text = PUSH;
expression(And);
*++text = XOR;
expr_type = INT;
}
```

其它的我们便不再详述。但这当中还有一个问题，就是指针的加减。在 C 语言中，指针加上数值等于将指针移位，且根据不同的类型移动的位移不同。如 `a + 1` ，如果 `a` 是 `char *` 型，则移动一字节，而如果 `a` 是 `int *` 型，则移动 4

个字节（32位系统）。

另外，在作指针减法时，如果是两个指针相减（相同类型），则结果是两个指针间隔的元素个数。因此要有特殊的处理。

下面以加法为例，对应的汇编代码为：

```
<expr1> + <expr2>

normal         pointer

<expr1>        <expr1>
PUSH           PUSH
<expr2>        <expr2>      |
ADD            PUSH         | <expr2> * <unit>
               IMM <unit>   |
               MUL          |
               ADD
```

即当 `<expr1>` 是指针时，要根据它的类型放大 `<expr2>` 的值，因此对应的源码如下：

```
else if (token == Add) {
// add
match(Add);
*++text = PUSH;
expression(Mul);

expr_type = tmp;
if (expr_type > PTR) {
// pointer type, and not `char *`
*++text = PUSH;
*++text = IMM;
*++text = sizeof(int);
*++text = MUL;
}
*++text = ADD;
}
```

相应的减法的代码就不贴了，可以自己实现看看，也可以看文末给出的链接。

自增自减

这次是后缀形式的，即 `p++` 或 `p--` 。与前缀形式不同的是，在执行自增自减后， `ax` 上需要保留原来的值。所以我们首先执行类似前缀自增自减的操作，再将 `ax` 中的值执行减/增的操作。

```
// 前缀形式 生成汇编代码
*++text = PUSH;
*++text = IMM;
*++text = (expr_type > PTR) ? sizeof(int) : sizeof(char);
*++text = (tmp == Inc) ? ADD : SUB;
*++text = (expr_type == CHAR) ? SC : SI;

// 后缀形式 生成汇编代码
*++text = PUSH;
*++text = IMM;
*++text = (expr_type > PTR) ? sizeof(int) : sizeof(char);
*++text = (token == Inc) ? ADD : SUB;
*++text = (expr_type == CHAR) ? SC : SI;
*++text = PUSH; //
*++text = IMM; // 执行相反的增/减操作
*++text = (expr_type > PTR) ? sizeof(int) : sizeof(char); //
*++text = (token == Inc) ? SUB : ADD; //
```

数组取值操作

在学习 C 语言的时候你可能已经知道了，诸如 `a[10]` 的操作等价于 `*(a + 10)` 。因此我们要做的就是生成类似的汇编代码：

```
else if (token == Brak) {
// array access var[xx]
match(Brak);
*++text = PUSH;
expression(Assign);
match(']');

if (tmp > PTR) {
// pointer, `not char *`
*++text = PUSH;
*++text = IMM;
*++text = sizeof(int);
*++text = MUL;
}
else if (tmp < PTR) {
printf("%d: pointer type expected\n", line);
exit(-1);
}
expr_type = tmp - PTR;
*++text = ADD;
*++text = (expr_type == CHAR) ? LC : LI;
}
```

代码

除了上述对表达式的解析外，我们还需要初始化虚拟机的栈，我们可以正确调用 `main` 函数，且当 `main` 函数结束时退出进程。

```
int *tmp;
// setup stack
sp = (int *)((int)stack + poolsize);
*--sp = EXIT; // call exit if main returns
*--sp = PUSH; tmp = sp;
*--sp = argc;
*--sp = (int)argv;
*--sp = (int)tmp;
```

当然，最后要注意的一点是：所有的变量定义必须放在语句之前。

本章的代码可以在 Github 上下载，也可以直接 clone

```
git clone -b step-6 https://github.com/lotabout/write-a-C-interpret
```

通过 `gcc -o xc-tutor xc-tutor.c` 进行编译。并执行 `./xc-tutor hello.c` 查看结果。

正如我们保证的那样，我们的代码是自举的，能自己编译自己，所以你可以执行 `./xc-tutor xc-tutor.c hello.c` 。可以看到和之前有同样的输出。

小结

本章我们进行了最后的解析，解析表达式。本章有两个难点：

1. 如何通过递归调用 `expression` 来实现运算符的优先级。
2. 如何为每个运算符生成对应的汇编代码。

尽管代码看起来比较简单（虽然多），但其中用到的原理还是需要仔细推敲的。

最后，恭喜你！通过一步步的学习，自己实现了一个C语言的编译器（好吧，是解释器）。

手把手教你做一个 C 语言编译器（9）：总结

恭喜你完成了自己的 C 语言编译器，本章中我们发一发牢骚，说一说编写编译器值得注意的一些问题；编写编译器时遇到的一些难题。

本系列：

1. 手把手教你做一个 C 语言编译器（0）：前言
2. 手把手教你做一个 C 语言编译器（1）：设计
3. 手把手教你做一个 C 语言编译器（2）：虚拟机
4. 手把手教你做一个 C 语言编译器（3）：词法分析器
5. 手把手教你做一个 C 语言编译器（4）：递归下降
6. 手把手教你做一个 C 语言编译器（5）：变量定义
7. 手把手教你做一个 C 语言编译器（6）：函数定义
8. 手把手教你做一个 C 语言编译器（7）：语句
9. 手把手教你做一个 C 语言编译器（8）：表达式

虚拟机与目标代码

整个系列的一开始，我们就着手虚拟机的实现。不知道你是否有同感，这部分对于整个编译器的编写其实是十分重要的。我认为至少占了重要程度的50%。

这里要说明这样一个观点，学习编译原理时常常着眼于词法分析和语法分析，而忽略了同样重要的代码生成。对于学习或考试而言或许可以，但实际编译项目时，最为重要的是能“跑起来”，所以我们需要给予代码生成高度的重视。

同时我们也看到，在后期解析语句和表达式时，难点已经不再是语法分析了，而是如何为运算符生成相应的汇编代码。

词法分析

我们用了很暴力的手段编写了我们的词法分析器，我认为这并无不可。

但你依旧可以学习相关的知识，了解自动生成词法分析器的原理，它涉及到了“正则表达式”，“状态机”等等知识。相信这部分的知识能够很大程度上提高你的编程水平。

同时，如果今后你仍然想编写编译器，不妨试试这些自动生成工具。

语法分析

长期以来，语法分析对我而言一直是迷一样的存在，直到真正用递归下降的方式实现了一个。

我们用了专门的一章讲解了“递归下降”与 BNF 文法的关系。希望能减少你对理论的厌恶。至少，实现起来并不是太难。

如果有兴趣，可以学习学习这些文法，因为已经有许多自动生成的工具支持它们。这样你就不需要重复造轮子。可以看看 yacc 等工具，更先进的版本是 `bsion` 。同时其它语言也有许多类似的支持。

题外话，最近知道了一个叫“PEG 文法”的表示方法，无论是读起来，还是实现起来，都比 BNF 要容易，你也可以学习看看。

关于编代码

这也是我自己的感慨吧。无论多好的教程，想要完全理解它，最好的方式恐怕还是要自己实现它。

只是在编写代码的过程中，我们会遇到许多的挫折，例如需要考虑许多细节，或是调试起来十分困难。但也只有真正静下心来去克服它，我们才能有所成长吧。

例如在编写表达式的解析时，大量重复的代码特别让人崩溃。还有就是调试编译器，简直痛苦地无话可说。

P.S. 如果你按这个系列自己编写代码，记得事先写一些用于输出汇编代码的函数，很有帮助的。

还有就是写这个系列的文章，开始的冲动过了之后，每写一篇都特别心烦，希望文章本身没有受我的这种情绪影响吧。

结语

编程有趣又无趣，只有身在其中的我们才能体会吧。